FIAP ON

DESENVOLVIMENTO DE APPS
– PARTE 1 (ANDROID)

A LINGUAGEM *KOTLIN*

ROBERTO RODRIGUES JUNIOR

1

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

1 A LINGUAGEM KOTLIN

Antes de você criar aplicativos para o sistema operacional móvel que mais cresce no mundo, é necessário aprender a "falar a língua" dele. Neste capítulo, vamos trabalhar com a linguagem de programação mais recente, o Kotlin.c.

1.1 Apresentação

Kotlin é uma linguagem de programação criada em 2011 pela JetBrains, empresa conhecida pelas IDEs (*Integrated Development Environme*nt) comercializadas. Essa linguagem recebeu o nome de uma ilha russa situada próximo à costa de São Petersburgo, onde a equipe Kotlin reside.

Em julho de 2011, a JetBrains revelou que estava trabalhando em uma nova linguagem (Kotlin), com o objetivo de suprir características que não eram encontradas em outras. No ano seguinte, o projeto Kotlin foi colocado sob a licença Apache 2 de código aberto. Porém, apenas em 2016, foi lançada a primeira versão estável (KOTLIN, 2018).

Aos poucos, ela foi conquistando espaço na comunidade de desenvolvedores de aplicativos Android devido à sua facilidade de aprendizagem e interoperabilidade com a linguagem de programação Java.

Contudo, a grande notícia veio em 2017, no Google I/O 17, quando os engenheiros do Google Android anunciaram a Kotlin como a mais nova linguagem oficial da plataforma (KOTLIN, 2018).

1.2 Principais características

Como mostrado anteriormente, uma característica importante do Kotlin é a compatibilidade com o Java. Assim, todas as APIs compatíveis com o Java também são compatíveis para o Kotlin. É possível destacar outras características, tais como:

- Aplicativos em Kotlin possuem desempenho equivalente a aplicativos desenvolvidos em Java.

- É possível realizar a declaração de variáveis de forma implícita (sem explicitar o tipo de dado, por exemplo). Essa característica permite a adaptação de desenvolvedores advindos de abordagens funcionais (Python, JavaScript, Swift e similares).
- A linguagem é considerada *type-safe*, como o Java, e também *null-safe*, ou seja, caso necessite que uma variável receba valor nulo **(null)**, terá de definir isso de forma explícita;
- Utiliza *type casts* que asseguram o desenvolvedor nas diferentes conversões e comparações entre tipos.

1.3 Por que desenvolver para Android com Kotlin?

Dadas as suas características de uma linguagem multiparagima, ela pode ser uma opção para o desenvolvimento dos aplicativos Android, trazendo as vantagens de uma linguagem moderna sem apresentar restrições em relação ao Java. Em adicional ao já apresentado em relação ao Kotlin, podem ser destacados:

- **Compatibilidade**: Kotlin é totalmente compatível com a JDK (*Java Devlopment Kit*). Além disso, oferece o suporte legado para versões anteriores do Android.
- **Desempenho**: um aplicativo Kotlin pode ser equivalente a um desenvolvido em Java, gerando *bytecodes* semelhantes. Devido ao Kotlin orientar o desenvolvedor a códigos enxutos, existe a possiblidade da geração de *bytecodes* otimizados.
- **Curva de aprendizado**: a curva de aprendizado para Kotlin é favorável a desenvolvedores de outras linguagens de programação. Para desenvolvedores em Java, existem ferramentas para a migração segura de código-fonte para integração com Kotlin.

1.4 REPL (Read-Eval-Print Loop)

1.4.1 Ambiente de estudos

Para possibilitar o estudo na linguagem de programação Kotlin, é sugerido um ambiente de estudos no formato REPL (*Read-Eval-PrintLoop*) para escrever programas simples e que reforcem o aprendizado de elementos essenciais da linguagem. A sugestão é utilizar o portal (https://play.kotlinlang.org) para teste dos exemplos e incremento gradativo do aprendizado.

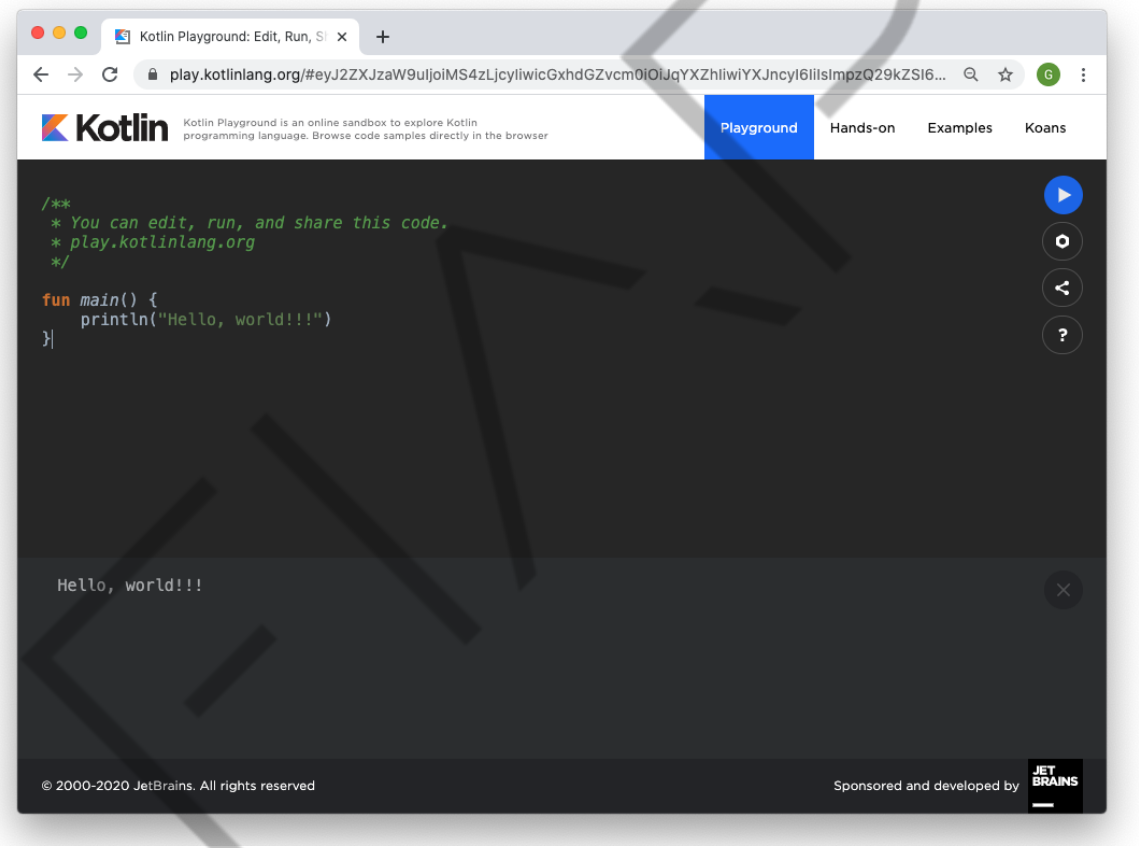

Figura 1.1 – Portal para teste de trechos de código em Kotlin
Fonte: Kotlin (2020)

Para executar os exemplos desenvolvidos neste material, devem ser feitas as seguintes observações:

- Na parte superior da Figura “Portal para teste de trechos de código em Kotlin”, podemos selecionar a opção do menu *Examples*, essa opção contém exemplos que poderão ser utilizados para facilitar o aprendizado da linguagem.

- Na parte central da Figura “Portal para teste de trechos de código em Kotlin” é o local destinado à digitação dos exemplos. Inicialmente, essa digitação é distribuída nas seguintes partes:
  - Nas linhas 1 a 4 foram inseridos comentários, o que será explicado futuramente. Manter os comentários é opcional.
  - Nas linhas 6 e 8 se encontra a função principal de execução dos blocos de códigos que serão digitados (*fun main*). É importante lembrar que a assinatura função “main” não deve ser alterada.
  - A linha 7 contém a função *println* com o texto entre aspas *Hello World*. A função *println*, assim como na linguagem Java, permite gerar a saída do programa em terminal (localizado na parte inferior da Figura “Portal para teste de trechos de código em Kotlin”).
  - No canto superior direito, existe uma seta na cor verde em conjunto com a palavra *Run*. Ao clicar nesse botão, o código-fonte será executado. Em caso de inconsistência no código-fonte, serão fornecidas mensagens para auxiliar na correção.

Na parte inferior da Figura “Portal para teste de trechos de código em Kotlin”, vemos a *debug area*, em que aparece a mensagem “Hello, world!!!”. Caso nosso código tenha algum erro, é na *debug area* que temos mais informações sobre esse erro.

1.5 Comentários e variáveis

1.5.1 Comentários

Toda linguagem de programação possui um recurso que permite que o desenvolvedor possa deixar um comentário, seja para documentar um trecho de código ou para lembrá-lo de revisar algo que tenha feito. Também são utilizados comentários quando se deseja que certo trecho do código-fonte não seja interpretado. Em Kotlin, existem dois tipos de comentários: em única linha ou em múltiplas linhas. Para criarmos um comentário em única linha, utilizam-se os caracteres `//`, e para comentários de múltiplas linhas, iniciamos o comentário com `/*` e finalizamos com `*/`.

O Código-fonte “Comentários” apresenta exemplos do uso de comentário em única linha e em múltiplas linhas.

```
// Comentários em uma linha
// Quantidade de alunos por sala de aula

var sala1 = 20
var sala2 = 15
var sala3 = 18

var total = sala1 + sala2 + sala3
println(total)

 /*
  * Comentários em bloco.
  * Podemos digitar várias explicações em
  * um mesmo bloco de comentários
  */

sala1 = 25
sala2 = 27
sala3 = 34

total = sala1 + sala2 + sala3
println(total)
```

Código-fonte 1.1 – Comentários
Fonte: Elaborado pelo autor (2019)

Nesse caso, o resultado impresso é, inicialmente, 53, que é a soma dos conteúdos das variáveis `sala1`(20), `sala2`(15) e `sala3`(18). Depois, as variáveis recebem novas atribuições (25, 27 e 24, respectivamente) e o valor da soma é novamente atribuída à variável `total`, resultando no valor 86. A declaração e atribuição em variáveis são detalhadas na seção a seguir.

1.5.2 Variáveis e constantes

A declaração de variáveis em Kotlin é feita por meio do uso da palavra reservada `var`**,** que indica o início da declaração de um objeto cujo valor pode mudar ao longo do código (objeto mutável, ou variável). A declaração da variável pode ser explícita (com a informação do tipo de dados ou objeto) ou implícita (na qual o interpretador definirá a partir do conteúdo a ser inicialmente atribuído).

Para declarar uma variável explícita, é utilizada, inicialmente, a palavra reservada `var`, seguindo do nome da variável, dois pontos e o seu tipo.

Uma das características da linguagem Kotlin é a existência da **Inferência de Tipo**, ou seja, não é obrigatória a definição do tipo de uma variável, caso ela seja inicializada na declaração. Vejamos o Código-fonte “Variáveis”:

```
//Criando variável com tipo explícito
var faculdade: String

//Criando variável utilizando inferência de tipo
var cidade = "São Paulo"
```

Código-fonte 1.2 – Variáveis
Fonte: Elaborado pelo autor (2019)

No exemplo do Código-fonte “Exemplo com uso de declaração implícita”, a inicialização das variáveis com conteúdo do tipo String resulta no ajuste de tipo por parte do interpretador, garantindo a manutenção do mesmo tipo em todo o código-fonte (*type safe*).

```
//Usando lowerCamelCase. Cada nova palavra, começa em
maiúscula
var firstName = "Luke"
var lastName = "Skywalker"
println (firstName)
println (lastName)
```

Código-fonte 1.3 – Exemplo com uso de declaração implícita
Fonte: Elaborado pelo autor (2019)

Para a criação de constantes, ou seja, objetos que não modificam seu valor ao longo do código (objetos imutáveis), é utilizada a palavra reservada `val`. O Código-fonte “Criando constantes” mostra exemplos por meio das constantes `pi` e `gravity`. Quando se tenta atribuir novo valor à constante `gravity`, é gerado erro pelo interpretador.

```
//Criando constantes
val pi = 3.141592
val gravity = 9.81
```

```
    //Note que não é possível alterar o valor de uma
constante
    gravity = 10.01  //ERRO
```

Código-fonte 1.4 – Criando constantes
Fonte: Elaborado pelo autor (2019)

DICA: Sempre procure utilizar constantes em seu código, a menos que realmente precise modificar o objeto futuramente. O acesso às constantes na memória é mais rápido do que o acesso às variáveis.

1.6 Tipos

Existem tipos pré-definidos em Kotlin que são utilizados com frequência, tais como: representações para números inteiros, números com casas decimais, valores booleanos, dentre outros. A Tabela “Tipos pré-definidos” lista os tipos de dados comumente utilizados.

Tabela 1.1 – Tipos predefinidos

Tipo	Tamanho em bits
Long	64
Int	32
Short	16
Byte	8
Double	64
Float	32

Fonte: Elaborado pelo autor (2019)

1.6.1 Tipos Inteiros (Long, Int, Short e Byte)

Quando é necessário representar, em Kotlin, um número que não possui casas decimais, as representações para números inteiros podem ser utilizadas de acordo com a necessidade e decisões de projeto de software. O tipo **Long** utiliza 64 bits de memória para ser representado, ou seja, pode variar entre -9223372036854775808 e 9223372036854775807. O tipo **Int** utiliza 32 bits de memória e pode variar entre -

2.147.483.648 a 2.147.483.647. O tipo **Short** utiliza 16 bits de memória e pode variar entre -32768 a 32767 por fim, o tipo Byte utiliza 8 bits de memória e pode variar entre -128 a 127. Cada tipo pode ser aplicado a diferentes situações de projeto. O Código-fonte “Utilizando Inteiros” apresenta os valores MAX_VALUE e MIN_VALUE, os quais retornam, respectivamente, a faixa de valor máximo e mínimo para cada tipo de dado suportado. Além disso, é apresentado um exemplo da declaração de variável em modo implícito e modo explícito.

```
    var value1 = 500       //Aqui, a inferência é para Int
    var value2: Int = 500  //Declaração explícita

    // Apresentando o value1
    println(value1)
    // Apresentando o value2
    println(value2)

    //Forma de mostrar o valor máximo aceito pelo tipo
    println(Int.MAX_VALUE)

    //Forma de mostrar o valor mínimo aceito pelo tipo
    println(Int.MIN_VALUE)
```

Código-fonte 1.5 – Utilizando Inteiros
Fonte: Elaborado pelo autor (2019)

1.6.2 Double e Float (Números com casas decimais)

Para representar números que possuem casas decimais, o **Double** (que é o tipo inferido automaticamente quando atribuímos um número com casas decimais à uma variável) e o **Float** podem ser utilizados. A diferença entre *Double e Float* é que *Double* ocupa 64 bits na memória, podendo trabalhar com números maiores, e *Float* utiliza 32 bits de memória. O Código-fonte “Utilizando Inteiros” mostra exemplos da utilização dos tipos Double e Float.

```
    //Tipo Double é o tipo padrão para valores numéricos com
casas decimais

    var balance = 1500.75          //Double inferido
automaticamente
    var sallary: Double = 1200.50  //Double explícito
```

```
    //Para usarmos Float, precisamos adicionar a letra F
maiúscula ou minúscula no final da informação
    var height = 1.82f
    var temperature: Float = 35.9F

    //Apresentando as informações
    println(balance)
    println(sallary)

    println(height)
    println(temperature)
```

Código-fonte 1.6 – Utilizando Double e Float
Fonte: Elaborado pelo autor (2019)

1.6.3 String e Char

Blocos de texto são representados em Kotlin pelo tipo String, que é definido por um ou mais caracteres entre aspas. Esse é o tipo inferido quando se atribui um texto a uma variável. É possível utilizar o tipo **Char** quando for necessário ocupar o espaço de apenas um único caractere, porém vale ressaltar que, para atribuir um **Char** a uma variável, é necessário definir de forma explícita o tipo **Char** na variável, pois, do contrário, será *String,* mesmo que o texto contenha apenas um caractere. Outro detalhe que é importante lembrar é que o tipo **Char** necessita que o caractere esteja entre aspas simples. O Código-fonte “String e Character” apresenta exemplos do uso da String e do Char, além de mostrar a operação de *casting* de Char a para String utilizando o operador “$”.

```
    var module: String = "Introdução ao Kotlin"
    var schoolName = "FIAP"

    //Note que letter, na linha abaixo, é uma String
    //Devido à inferência de tipo
    var letter = "A"

    //Para usarmos Char, precisamos definir explicitamente
    var gender: Char = 'M'

    // Apresentando os valores
    println(module)
    println(schoolName)
    println(letter)

    // Para apresentação de valores Char é necessário a
conversão para String
```

```
    //Utilize o caracter $ para a saída. Explicaremos o uso
do $ nas próximas páginas
    println("$gender")
```

Código-fonte 1.7 – String e Character
Fonte: Elaborado pelo autor (2019)

Quando é necessário inserir, em uma *String*, algum caractere reservado pela linguagem (por exemplo, o caractere **"**que define o início/fim de uma *String*), utilizamos o caractere \. Por meio dele, podem-se utilizar caracteres reservados, bem como outros conjuntos especiais, por exemplo: o *carriage return* (simulando a tecla ENTER), uma tabulação, entre outros.

```
    var text = "Este texto \"quebra\" em \nduas linhas"
    //Resultado:
    //Este texto "quebra" em
    //duas linhas

    //O \t gera uma tabulação
    var text2 = "Nota:\t 10"
    //Resultado:
    //Nota:     10

    // Apresentando os valores das variáveis
    println(text)
    println(text2)
```

Código-fonte 1.8 – Caractere especial $
Fonte: Elaborado pelo autor (2019)

Em Kotlin, é possível criar *Strings* utilizando uma técnica chamada *Interpolação de Strings*. É muito comum a atribuição de uma *String* à combinação de várias variáveis. Por exemplo, supondo que tenhamos uma variável que represente a nota do aluno (`studentGrade`), o nome (`studentName`) e o resultado de sua avaliação (aprovado ou reprovado), podemos criar uma variável chamada `message`, que conterá uma *String* informando todos os dados dos alunos e a sua avaliação. Ex.: "O aluno João tirou 8.5 e está aprovado". Para criarmos essa *String*, é necessário juntar informações de três variáveis e, para isso, utiliza-se a técnica de interpolação.

Isso é feito colocando, dentro da *String* principal, as variáveis dentro de parênteses, precedido pelo caractere **$.** Veja no exemplo do Código-fonte "Interpolação de Strings":

```
    val studentGrade = 8.5
    val studentName = "João"
    val result = "aprovado"

    val message = "O aluno $studentName tirou $studentGrade
e está $result"
    println(message)
    //Resultado:
    //O aluno João tirou 8.5 e está aprovado
```

Código-fonte 1.9 – Interpolação de Strings
Fonte: Elaborado pelo autor (2019)

1.6.4 Bool (Booleanos)

Booleanos são tipo simples, que ocupam apenas 1 bit de memória e apenas aceitam dois estados, 0 ou 1. Em Kotlin, booleanos são definidos pelo tipo *Boolean* e aceitam **true** (verdadeiro) ou **false** (falso), conforme exemplo do Código-fonte "Booleanos".

```
    var isApproved = true
    var firstTime: Boolean = false
```

Código-fonte 1.10 – Booleanos
Fonte: Elaborado pelo autor (2019)

1.6.5 Pair (Par)

Existe, em Kotlin, um tipo muito útil, em determinados casos, para compor pares de dados. Um dos casos mais utilizados é quando, por exemplo, precisamos definir que uma função precisa retornar mais de um tipo ao mesmo tempo. O ***Pair*** é um tipo composto formado por dois valores. O Código-fonte "Endereço em String" mostra um exemplo do uso do Pair.

```
    val (address,number) = Pair("Av. Lins de Vasconcelos",
1264)
    println(address)
    println(number)
```

Código-fonte 1.11 – Endereço em String
Fonte: Elaborado pelo autor (2019)

1.6.6 Tipo Nullable (null safety)

Por padrão, variáveis em Kotlin não aceitam o uso de valor nulo, tendo, o desenvolvedor, que realizar as condições de contorno necessárias. Todavia, o recurso da atribuição de valor nulo a uma variável pode ser utilizado, sendo essa abordagem denominada *null safety* pela comunidade dos programadores.

O sistema de tipos de Kotlin é avançado o suficiente a ponto de poder monitorar a diferença entre tipos *nullable* e não *nullable.* A atribuição de valor nulo a uma variável só pode ser realizada quando a sua declaração seguir com o sufixo “?”, como apresenta o Código-fonte “Trabalhando com null safety”.

```
     //A linha abaixo o código não irá compilar
     //var driverLicense: String = null

     // A atribuição de null a um var também não será
compilada
     //var driverLicense: String = "6789877"
     //driverLicense = null // não compila

     // Para que uma variável contenha um valor null é
necessário
     // o uso do sufixo ? no tipo
     var driverLicense: String? = null
     println(driverLicense)

     driverLicense = "6789877"
     println(driverLicense)

     driverLicense = null // agora compila
     println(driverLicense)
```

Código-fonte 1.12 – Trabalhando com null safety
Fonte: Elaborado pelo autor (2019)

Por sua vez, o Código-fonte “Trabalhando com null safety (Teste de Tipo Nulo)” apresenta um exemplo de como a verificação de uma variável nullable possui valor nulo.

```
     var driverLicense: String? = null
     //driverLicense = "6789877"
     if (driverLicense != null) {
         println("A carteira de motorista é $driverLicense")
     } else {
         println("Esta pessoa não possui carteira de
motorista")
```

```
    }

    //Resultado: Esta pessoa não possui carteira de
motorista
```

Código-fonte 1.13 – Trabalhando com null safety (Teste de Tipo Nulo)
Fonte: Elaborado pelo autor (2019)

1.7 Coleções

Muitas das linguagens de programação contêm recursos para o uso de coleções de dados. As principais coleções em Kotlin são *Array, List, Set e Map.*

1.7.1 Array

Uma das principais coleções em Kotlin é o *Array,* que é uma coleção ordenada de elementos de mesmo tipo, ou seja, não é possível misturar tipos dentro de um *Array.* Sendo assim, se definimos um *Array* de ***Int,*** todos os elementos dessa coleção devem necessariamente ser **Int,** não podendo ter um *String, Double*, ou qualquer outro tipo. O Código-fonte “Arrays” apresenta algumas formas de criarmos *Array* em Kotlin.

```
    //Criando um Array de Strings vazio
    var emptyArray =  arrayOf<String>()

    //Criando um Array de Strings e alimentando valores na
criação
    var shoppingList = arrayOf<String>("Leite", "Pão",
"Manteiga", "Açúcar")

    //Usando inferência
    var inferredShoppingList = arrayOf("Leite", "Pão",
"Manteiga", "Açúcar")

    //Testando se um Array está vazio
    if (shoppingList.isEmpty()) {
        println("A lista de compras está vazia")
    } else {
        println("A lista de compras NÃO está vazia")
    }

    //Recuperando o total de elementos do Array
    println("Nossa lista de compras possui
${shoppingList.size} itens")
    //Resultado: Nossa lista de compras possui 4 itens
```

Código-fonte 1.14 – Arrays
Fonte: Elaborado pelo autor (2019)

Veja que o *Array* possui uma função chamada *isEmpty()*, que é um *Bool* que nos informa se o *array* está ou não vazio, e também possui uma propriedade que retorna o total de itens, a propriedade *size.* Essas propriedades estão presentes em todas as coleções usadas em Kotlin.

DICA: Em Kotlin, uma String também é uma coleção de caracteres, ou seja, se quisermos saber o total de letras presentes em uma String, basta utilizarmos a função count(). Faça o teste!

Para recuperar um elemento de um *Array*, esse deve ser acessado pelo uso de *subscript*, ou seja, definindo, entre colchetes, o índice no qual se encontra esse elemento dentro do *Array*. Ele pode ser usado para modificar um elemento de um *Array*. Veja o exemplo do Código-fonte “Acessando itens de um Array”:

```
     //Criando um Array de Strings vazio
     var emptyArray =  arrayOf<String>()

     //Criando um Array de Strings e alimentando valores na
criação
     var shoppingList = arrayOf<String>("Leite", "Pão",
"Manteiga", "Açúcar")

     //Usando inferência
     var inferredShoppingList = arrayOf("Leite", "Pão",
"Manteiga", "Açúcar")

     //Testando se um Array está vazio
     if (shoppingList.isEmpty()) {
        println("A lista de compras está vazia")
     } else {
        println("A lista de compras NÃO está vazia")
     }

     //Recuperando o total de elementos do Array
     println("Nossa lista de compras possui
${shoppingList.size} itens")
     //Resultado: Nossa lista de compras possui 4 itens

     println("Listando todos os itens:")
     println(shoppingList[0]) //Primeiro item do array
     println(shoppingList[1]) //Segundo item Acessando itens
de um Array do array
     println(shoppingList[2]) //Terceiro item do array
     println(shoppingList[3]) //Quarto item do array
```

Código-fonte 1.15 – Acessando itens de um Array
Fonte: Elaborado pelo autor (2019)

1.7.2 List

List é uma coleção muito versátil, pois permite elementos repetidos; é ideal para quando precisarmos definir um conjunto de itens cujo valor poderá repetir, como a lista dos produtos que estão em um supermercado ou a lista de itens em um carrinho de compras, por exemplo. Para criar um *List,* é necessário definir o tipo explicitamente, pois a inferência, nesse caso, atribuiria conteúdo a um *Array*. Um List deve ser criado usando a palavra *ArrayList* seguida do tipo, entre os sinais < e >.

Vamos aprender a trabalhar com *List* por meio de alguns exemplos listados no Código-fonte “List”.

```
     //Criando um List de Strings
     var movies = ArrayList<String> ()
        movies.addAll(listOf(
        "Matrix",
        "Vingadores",
        "Jurassic Park",
        "De Volta para o Futuro"
        ))

     //Criando um list vazio
     var movies2 = ArrayList<String> ()

     //Inserindo elementos
     movies.add("Homem-Aranha: De Volta para o Lar")
     println(movies.count()) //5
     println("\n")

     //Perceba que o código abaixo irá alterar a quantidade
     //de itens do List pois ele aceita itens repetidos.
     movies.add("Homem-Aranha: De Volta para o Lar")
     println(movies)   //[Matrix, Vingadores, Jurassic Park,
De Volta para o Futuro, Homem-Aranha: De Volta para o Lar,
Homem-Aranha: De Volta para o Lar]
     println(movies.count()) //6 ( 2 elementos repetidos )
     println("\n")


     //Removendo 2 elementos repetidos
     movies.remove("Homem-Aranha: De Volta para o Lar")
     movies.remove("Homem-Aranha: De Volta para o Lar")

     println(movies)   //["Vingadores", "De Volta para o
Futuro", "Matrix", "Jurassic Park"]
     println("\n")
```

```
     //Percorrendo um List
     for (movie in movies) {
        println(movie)
     }
     println("\n")

     //Verificando se determinado elemento está contido no
List
     if (movies.contains("Matrix")) {
        println("Matrix está na minha lista de filmes
favoritos!!")
     }
     println("\n")

     //Vamos criar um novo List para realizarmos algumas
operações
     //No exemplo abaixo usaremos um formato mais
simplificado de criação de List
     var myWifeMovies  = listOf(
        "De Repente 30",
        "Mensagem para você",
        "Sintonia de Amor",
        "De Volta para o Futuro",
        "Jurassic Park"
     )


     //Criando um List com todos os filmes
     var allMovies = movies + myWifeMovies
     println(allMovies)
     //[Matrix, Vingadores, Jurassic Park, De Volta para o
Futuro, De Repente 30, Mensagem para você, Sintonia de Amor,
De Volta para o Futuro, Jurassic Park]
     println("\n")
```

Código-fonte 1.16 – List
Fonte: Elaborado pelo autor (2019)

1.7.3 Set

Set é uma coleção versátil, pois ela não permite elementos repetidos, é ideal para quando for necessário definir um conjunto de itens cujo valor não pode se repetir, como os alunos que estão em uma turma ou a nossa lista de filmes favoritos, por exemplo. Para criar um *Set*, precisamos definir o tipo explicitamente, pois a inferência, nesse caso, atribuiria conteúdo a um *Array* por padrão. Um *Set* pode ser criado usando a palavra *HashSet* seguida do tipo, entre os sinais < e >. O Código-fonte “Dicionários” apresenta exemplos da utilização do Set.

```
    //Criando um Set de Strings
    var movies = HashSet<String> ()

    var catalog = listOf(
           "Matrix",
           "Vingadores",
           "Jurassic Park",
           "De Volta para o Futuro"
    )

    movies.addAll(catalog)
    //Perceba que o catálogo de filmes está ordenado
alfabeticamente
    println(movies) //[Jurassic Park, Matrix, De Volta para
o Futuro, Vingadores]
    println(movies.count()) //4
    println("\n")

    //Criando um set vazio
    var movies2 = HashSet<String> ()

    //Inserindo elementos
    movies.add("Homem-Aranha: De Volta para o Lar")
    println(movies) //[Homem-Aranha: De Volta para o Lar,
Jurassic Park, Matrix, De Volta para o Futuro, Vingadores]
    println(movies.count()) //5
    println("\n")

    //Perceba que o código abaixo NÃO irá alterar a
quantidade
    //de itens do Set pois ele NÃO aceita itens repetidos.
    movies.add("Homem-Aranha: De Volta para o Lar")
    println(movies)   //[Homem-Aranha: De Volta para o Lar,
Jurassic Park, Matrix, De Volta para o Futuro, Vingadores]
    println(movies.count()) //5 ( Nenhum elemento repetido.
E ainda tudo ordenado )
    println("\n")

    //Removendo elemento
    movies.remove("Homem-Aranha: De Volta para o Lar")
    println(movies)   //["Vingadores", "De Volta para o
Futuro", "Matrix", "Jurassic Park"]
    println(movies.count()) //4
    println("\n")

    //Percorrendo um Set
    for (movie in movies) {
       println(movie)
    }
```

```
     println("\n")

     //Verificando se determinado elemento está contido no
List
     if (movies.contains("Matrix")) {
        println("Matrix está na minha lista de filmes
favoritos!!")
     }
     println("\n")

     //Vamos criar um novo set para realizarmos algumas
operações
     //No exemplo abaixo usaremos um formato mais
simplificado de criação de Set
     var myWifeMovies  = setOf(
        "De Repente 30",
        "Mensagem para você",
        "Sintonia de Amor",
        "De Volta para o Futuro",
        "Jurassic Park"
     )


     //Criando um Set com todos os filmes. SEM repetição.
TUDO ordenado ;)
     var allMovies = movies + myWifeMovies
     println(allMovies) // [Jurassic Park, Matrix, De Volta
para o Futuro, Vingadores, De Repente 30, Mensagem para você,
Sintonia de Amor]
     println(allMovies.count()) //7
```

Código-fonte 1.17 – Set
Fonte: Elaborado pelo autor (2019)

1.7.4 Map

Map é uma coleção organizada em pares <Chave, Valor> ou <Key, Value>. Ela possibilita a inserção de informações que tenham chaves únicas com seus valores (números, datas, textos etc.) relacionados. Essa coleção é muito utilizada para quando precisamos realizar pesquisas específicas por meio da chave ou até mesmo pelo valor. Para criar um *Map*, é necessário definir o tipo explicitamente. Um Map pode ser criado com a palavra *HashMap* seguida do tipo, entre os sinais < e >. O Código-fonte “Dicionários” apresenta exemplos do uso de Maps.

```
     //Criando um Map de Strings
     var movies = HashMap<Int,String> ()
     var catalog = mapOf(
            Pair(10,"Matrix"),
```

```
            Pair(20,"Vingadores"),
            Pair(30,"Jurassic Park"),
            Pair(40,"De Volta para o Futuro")
    )
    //Utilize o método putAll para inserir o catalogo
    movies.putAll(catalog)

    //Perceba que o catalogo de filmes está ordenado
alfabeticamente
    println(movies) //{40=De Volta para o Futuro, 10=Matrix,
20=Vingadores, 30=Jurassic Park}
    println(movies.count()) //4
    println("\n")

    //Criando um set vazio
    var movies2 = HashSet<String> ()

    //Inserindo 1 elemento
    movies.put(25,"Homem-Aranha: De Volta para o Lar")
    println(movies) //{40=De Volta para o Futuro, 25=Homem-
Aranha: De Volta para o Lar, 10=Matrix, 20=Vingadores,
30=Jurassic Park}
    println(movies.count()) //5
    println("\n")

    //Perceba que o código abaixo irá alterar a quantidade
    //de itens do Map pois ele aceita itens da chave NÃO
repetidos.
    //movies.put(25,"Homem-Aranha: De Volta para o Lar")

    //Faça um teste com a linha superior de código e a
inferir
    movies.put(35,"Homem-Aranha: De Volta para o Lar")

    println(movies)   //{40=De Volta para o Futuro,
25=Homem-Aranha: De Volta para o Lar, 10=Matrix, 35=Homem-
Aranha: De Volta para o Lar, 20=Vingadores, 30=Jurassic Park}
    println(movies.count()) //6
    println("\n")

    //Removendo elemento
    movies.remove(25)
    println(movies)   //{40=De Volta para o Futuro,
10=Matrix, 35=Homem-Aranha: De Volta para o Lar,
20=Vingadores, 30=Jurassic Park}
    println(movies.count()) //5
    println("\n")

    //Pecorrendo um Map
    for (movie in movies) {
       println(movie)
```

```
    }
    println("\n")

    //Verificando se determinado elemento está contido no
List
    if (movies.containsValue("Matrix")) {
       println("Matrix está na minha lista de filmes
favoritos!!")
    }
    println("\n")

    //Vamos criar um novo map para realizarmos algumas
operações
    //No exemplo abaixo usaremos um formato mais
simplificado de criação de Map
    var myWifeMovies  = mapOf(
       Pair(100,"De Repente 30"),
       Pair(200,"Mensagem para você"),
       Pair(300,"Sintonia de Amor"),
       Pair(400,"De Volta para o Futuro"),
       Pair(500,"Jurassic Park")
    )
    //Criando um Map com todos os filmes)
    var allMovies = movies + myWifeMovies
    println(allMovies) // {40=De Volta para o Futuro,
10=Matrix, 35=Homem-Aranha: De Volta para o Lar,
20=Vingadores, 30=Jurassic Park, 100=De Repente 30,
200=Mensagem para você, 300=Sintonia de Amor, 400=De Volta
para o Futuro, 500=Jurassic Park}
    println(allMovies.count()) //10
    println("\n")

    //Lendo Chave e Valor separadamente
    for(movie in allMovies) {
       println("Chave => Key => ${movie.key}")
       println("Valor => Value => ${movie.value}")

       var title = movie.value.toUpperCase()
       println("UpperCase => ${title}")

       title = movie.value.toLowerCase()
       println("LowerCase => ${title}")

       println("\n")
    }

    //Executando uma pesquisa diretamente na chave do Map
    var film1 = allMovies.get(400)   //400=De Volta para o
Futuro
    println("Title => ${film1}") // Retorna String. Retorna
o Título
```

```
    var film2 = allMovies.get(999)   //Não existe
    println("Title => ${film2}") // Retorna null

    //Verificando a possibilidade de testar antes de
imprimir
    var code = 1234 // Experimente trocar o código
    var film3 = allMovies.get(code)   //Não existe

    if(film3.isNullOrEmpty()) {
       println("\nFilme com o código $code não encontrado!")
    } else {
       println("\nTitle => ${film3}") // Retorna o título
    }
```

Código-fonte 1.18 – Dicionários
Fonte: Elaborado pelo autor (2019)

1.8 Operadores

A maioria dos operadores existentes nas diversas linguagens de programação também está disponível em Kotlin, atuando da mesma forma: os **operadores unários** (que atuam apenas em um operando), os **operadores binários** (atuam em dois operandos) e os **operadores ternários** (atua em três operandos).

1.8.1 Atribuição (=)

O sinal = serve para atribuir um valor a uma variável, como mostra o Código-fonte “Operador de atribuição”.

```
    var height: Double = 1.75
```

Código-fonte 1.19 – Operador de atribuição
Fonte: Elaborado pelo autor (2019)

1.8.2 Aritméticos (+, -, *, /, %)

Os operadores aritméticos são utilizados para a realização de operações aritméticas, como: soma, subtração, mudança de sinal, módulo etc.

```
    var a = 12
    var b = 3

    var sum = a + b              //15
```

```
var subtract = a - b          //9
var multiplication = a * b    //36
var division = a / b          //4
var módulus = a % b           //Resto da divisão: 0
var minusA = -a               //-12
```

Código-fonte 1.20 – Operadores de atribuição
Fonte: Elaborado pelo autor (2019)

1.8.3 Compostos (+=, -=, *=, /=, %=, ++, --)

Os operadores compostos são junções dos operadores aritméticos com o operador de atribuição, ou seja, efetuam a operação e atribuem o valor na variável ao mesmo tempo, conforme mostra o Código-fonte “Operadores compostos”.

```
var a = 2
var b = 3
var newValue = 5

newValue += a   //7
newValue -= b   //4
newValue *= a   //8
newValue /= a   //4
newValue %= b   //Resto da divisão: 1

newValue++ //incrementando 1
println(newValue)

newValue-- //decrementando 1
println(newValue)
```

Código-fonte 1.21 – Operadores compostos
Fonte: Elaborado pelo autor (2019)

1.8.4 Operadores Lógicos (&&, ||, !)

Esses operadores, por sua vez, executam operações lógicas, ou seja, sempre retornam verdadeiro ou falso. O Código-fonte “Operadores lógicos” apresentam exemplos da utilização.

```
var yes = true
var no = false

println(yes && no)    //false
```

```
    println(yes || no)    //true
    println(!yes)         //false
```

Código-fonte 1.22 – Operadores lógicos
Fonte: Elaborado pelo autor (2019)

1.8.5 Operadores de Comparação (>, <, >=, <=, ==, !=)

Os operadores de comparação são utilizados quando se pretende comparar valores, possuindo retorno verdadeiro ou falso. O Código-fonte “Operadores de comparação” apresenta exemplos de uso.

```
    var a = 12
    var b = 3
    var c = 7
    var d = 3

    println(a > b)   //true
    println(a < b)   //false
    println(b >= d)  //true
    println(a <= c)  //false
    println(b == d)  //true
    println(b != d)  //false
```

Código-fonte 1.23 – Operadores de comparação
Fonte: Elaborado pelo autor (2019)

1.8.6 Estrutura de decisão em mesma linha

Em Kotlin, é possível implementar estruturas decisão em uma mesma linha, como mostra o Código-fonte “Operadores ternário”. Com essa sintaxe, pode-se avaliar uma condição e atribuir um valor, caso a condição seja verdadeira, e outro valor, caso seja falsa.

```
    var grade = 7.5
    var result = if (grade > 7.0) "aprovado" else
"reprovado"
    println(result)   //aprovado
```

Código-fonte 1.24 – Operadores ternário
Fonte: Elaborado pelo autor (2019)

1.8.7 Coalescência nula (?:)

Em Kotlin, o operador ?: permite decidir pelo uso de uma atribuição de redundância, caso a variável analisada seja nula. O Código-fonte “Operador Coalescência nula” mostra exemplos do uso.

```
var age: Int? = null
var myAge = age ?: 0    //0
println(myAge)

age = 25
var newAge = age ?: 0   //25
println(newAge)
```

Código-fonte 1.25 – Operador Coalescência nula
Fonte: Elaborado pelo autor (2019)

1.8.8 Closed Range(..) e Half Closed Range (until)

Estes operadores criam um intervalo de valores. O **Closed Range** utiliza (..) para criar um intervalo aberto, indicando o valor inicial e o valor final. Por sua vez, o **Half Closed Range** utiliza a palavra reservada (until) para criar um intervalo entre o valor inicial e o valor imediatamente anterior ao valor final (intervalo aberto no início e fechado ao final). Veja o exemplo no Código-fonte “Operadores Closed Range e Half Closed Range”, no qual esses operadores são utilizados em uma estrutura de repetição `for`.

```
println("\nClosed Range ..")
var numbers = 1..10
for (number in numbers) {
    println(number)   //Imprime de 1 a 10
}

println("\nHalf Closed Range (until)")
var newNumbers = (1 until 10)
for (number in newNumbers) {
    println(number)   //Imprime de 1 a 9
}
```

Código-fonte 1.26 – Operadores Closed Range e Half Closed Range
Fonte: Elaborado pelo autor (2019)

1.9 Estruturas condicionais e de repetição

Toda linguagem precisa ter uma estrutura na qual se pode tomar uma decisão e agir de acordo com ela, ou seja, podemos definir o fluxo de nosso código baseado no resultado de uma análise. A essas estruturas chamamos de estruturas condicionais e, em Kotlin, a mais utilizadas é a estrutura `if else`. Outro recurso existente em toda linguagem de programação é a possibilidade de executarmos o mesmo trecho de código o número de vezes que for necessário, seja controlado por um intervalo específico ou até que uma condição seja alcançada.

1.9.1 If – else – else if

Para uma tomada de decisão, caso certa condição seja verdadeira, e outra, caso seja falsa, faz-se uso da estrutura `if – else – else if`**,** como apresentado no Código-fonte “If – else – else if”.

```
    var number = 11
    if (number % 2 == 0) {
        println("Ele é par")
    } else {
        println("Ele é ímpar")
    }

    var temperature = 18
    var climate = ""
    if (temperature <= 0) {
        climate = "Muito frio"
    } else if (temperature < 14) {
        climate = "Frio"
    } else if (temperature < 21) {
        climate = "Clima agradável"
    } else if (temperature < 30) {
        climate = "Um pouco quente"
    } else {
        climate = "Muuuito quente"
    }
    println("Temperatura:$temperature graus
Status:$climate")
```

Código-fonte 1.27 – If – else – else if
Fonte: Elaborado pelo autor (2019)

1.9.2 When

É possível observar, no exemplo do Código-fonte “If – else – else if” que, no caso da temperatura, precisamos fazer uso de vários `else if` para percorrermos todos os cenários existentes. Em situações como essa, a estrutura *when* (A palavra reservada `when` tem correlação com a palavra `switch` utilizada em outras linguagens de programação.) é mais adequada, pois ela foi criada especificamente para validar uma série de cenários possíveis para uma variável.

Um detalhe, em Kotlin, o `when` precisa ser exaurido, ou seja, deve contemplar todos os possíveis cenários para aquela variável que está sendo validada. Porém em situações nas quais o cenário é amplo, faz-se o uso da cláusula `else`, (`default` em outras linguagens) que é o cenário escolhido quando nenhum dos outros é verdadeiro. Vejamos em exemplos do Código-fonte “When”:

```
var number = 7
when (number % 2) {
    0 ->
    println("$number é par")
    else ->
    println("$number é ímpar")
}

//Exemplo com vários cenários no mesmo case
var letter = "z"
when (letter) {
    "a", "e", "i", "o", "u" ->
    println("vogal")
    else ->
    println("consoante")
}

//Exemplo com range de letras
when (letter) {
    in "a".."f" ->
    println("Você está na turma 1")
    in "g".."l" ->
    println("Você está na turma 2")
    in "m".."r" ->
    println("Você está na turma 3")
    else ->
    println("Você está na turma 4")
}
```

```
    //Range de números
    var speed = 33
    when (speed) {
        in 0 until 20 ->
        println("Primeira marcha")
        in 20 until 40 ->
        println("Segunda marcha")
        in 40 until 50 ->
        println("Terceira marcha")
        in 50 until 90 ->
        println("Quarta marcha")
        else ->
        println("Quinta marcha")
    }
```

Código-fonte 1.28 – When
Fonte: Elaborado pelo autor (2019)

1.9.3 While / do while

Essa estrutura de repetição é utilizada quando se deseja que certo trecho de código seja executado enquanto (`while`) uma condição seja verdadeira, ou seja, o laço será encerrado no momento em que a condição for falsa. A estrutura `do while` é semelhante, porém ela sempre executa o código uma vez antes de validar a condição. Veja o exemplo do Código-fonte “While, do while”.

```
    //Usando: while
    var life = 10
    while (life > 0) {
        println("O jogador está com $life vidas")
        life = life - 1
    }

    println("\n")

    //Usando: do while
    var tries = 0
    var diceNumber = 0
    do {
        tries += 1

        diceNumber = ((Math.random() * 6) + 1).toInt()

        println("Tentativa:$tries <-> Número Randomizado:
$diceNumber")
```

```
    } while (diceNumber != 6)

    println("\nVocê tirou 6 após $tries tentativas")
```

Código-fonte 1.29 – While, do while
Fonte: Elaborado pelo autor (2019)

No Código-fonte “While, do while” o `while` é utilizado para executar a operação de imprimir a quantidade de vidas do jogador enquanto for maior do que **0.** Note como a quantidade de vidas foi reduzida em cada *loop,* garantindo que, em um determinado momento, essa quantidade seja zerada e o laço seja interrompido.

No segundo exemplo, foi feito o uso do comando **(**`do while`**)**, pois desejamos que a operação seja executada pelo menos uma vez (jogada de um dado para ver se cai o número 6). Nesse código, utilizamos um método chamado *random* da classe *Math*, que gera um número aleatório que vai de **0** até o valor menos **1,** ou seja, no nosso caso, de **0** a **5,** e depois somar com **1** para que tenha um número de **1** a **6**. O retorno desse método também é um `double` e, por isso, o transformamos em `Int` **(**utilizando a função `toInt()`**)** para podermos atribuir esse valor à variável **diceNumber** que, por inferência, é do tipo `Int`**.**

1.9.4 For in

Sem dúvida, a estrutura de repetição mais utilizada em Kotlin é o `for in`**.** Por meio dessa estrutura, é possível iterar (percorrer) uma coleção e recuperar todos os seus valores, o que é ideal quando necessitamos percorrer um *array* ou uma coleção, por exemplo. O Código-fonte “For in” apresenta um exemplo da utilização.

```
    //Percorrendo um Array
    var students = arrayOf(
        "João Francisco",
        "Pedro Henrique",
        "Gustavo Oliveira",
        "Janaina Santos",
        "Francisco José"
    )
    for (student in students) {
        println("O aluno $student veio na aula de hoje!")
    }

    //Percorrendo uma sequência (range)
    for (day in 1..30) {
```

```
        println("Estou no dia $day")
    }

    //Note abaixo que uma String também é uma coleção
    var name = "FIAP"
    for (letter in name) {
        println(letter)
    }
    //Vejamos como percorrer uma coleção,
    //imprimindo sua chave e valor. Nesta coleção
    //a chave é String e o valor é Int
    var people = mapOf(
        (25 to "Paulo"),
        (18 to "Renata"),
        (33 to "Kleber"),
        (51 to "Roberto"),
        (36 to "Carol")
    )
    //A variável person, abaixo, recebe a chave
    //(key) e o valor (value) de cada elemento da coleção
    for (person in people) {
        println("${person.key} => ${person.value}")
    }

    //Podemos quebrar a execução de um laço usando
    //o comando break
    var grades = arrayOf(10.0, 9.0, 8.5, 7.0, 9.5, 5.0,
22.0, 6.5, 10.0)
    for (grade in grades) {
        println(grade)
        if (grade < 0.0 || grade > 10.0) {
            println("Nota inválida")
            break
        }
    }
```

Código-fonte 1.30 – For in
Fonte: Elaborado pelo autor (2019)

1.10 Enumeradores

Enumeradores (ou `enum`) são tipos criados pelo usuário que servem para se definir um tipo comum para um conjunto fechado de valores. São utilizados para cenários os quais devem armazenar uma informação baseada em um conjunto limitado de possibilidades. Para criarmos um *enum,* deve ser utilizada a palavra reservada **enum** em conjunto com a palavra *class*, seguida do nome do enumerados (com inicial maiúscula) e, entre chaves, definimos todos os valores possíveis. No

Código-fonte “Enum” é criado um enum que serve para definir uma bússola, com quatro possíveis valores (norte, sul, leste e oeste).

```
     //Definindo um enum fora da função main
     enum class Compass {
         north,
         east,
         west,
         south
     }

     fun main(args: Array<String>) {

         //Criando uma variável do tipo Compass
         var direction = Compass.north

         //Como Kotlin trabalha com inferência de tipo,
podemos usar
         // somente .valor, caso o tipo seja definido
explicitamente
         var direction2: Compass = Compass.south

         println("Minha direção é $direction")
         //Minha direção é north

         //Enums são muito usados com switch para análise do
valor
         when (direction) {
             Compass.north ->
                 println("Estamos indo para o norte")
             Compass.south ->
                 println("Estamos indo para o sul")
             Compass.east ->
                 println("Estamos indo para o leste")
             Compass.west ->
                 println("Estamos indo para o oeste")
         }
          //Estamos indo para o norte

          //Outra forma de apresentar informações de um Enum
         Compass.values().forEach {
             println(it)
         }
     }
```

Código-fonte 1.31 – Enum
Fonte: Elaborado pelo autor (2019)

É possível notar, no código anterior que, quando imprimimos o valor de um `enum`, ele imprime o próprio nome do valor (*north* no nosso exemplo). *Enums* são

muito utilizados com `When`, pois geralmente precisamos verificar qual valor ele possui, para tomarmos uma decisão.

1.10.1 Valores padrões

Em Kotlin, pode-se definir o tipo de um *enum* e, além disso, atribuir um valor padrão a cada um dos casos, conforme mostra o Código-fonte “Enum com valores padrões”.

```
     //Enum que define as posições das poltronas em um avião
     //Veja que é possível atribuir um valor padrão a cada
uma delas
     enum class SeatPosition(var seat: String) {
         aisle("corredor"),
         middle("meio"),
         window("janela")
     }

     //Enum de Int com valores padrões
     enum class Month(var m: Int) {
         january(1), february(2), march(3), april(4), may(5),
june(6),
         july(7), august(8), september(9), october(10),
november(11), december(12)
     }

     fun main(args: Array<String>) {

          var passengerSeat = SeatPosition.window

          //Para imprimir o valor padrão, usamos o nome
utilizado na construção do enum. Veja:
          println(passengerSeat.seat)   //janela

          var currentMonth: Month = Month.june
          println("Estamos no mês ${currentMonth.m} do ano")
     }
```

Código-fonte 1.32 – Enum com valores padrões
Fonte: Elaborado pelo autor (2019)

Nesse caso, ao criar o *enum SeatPosition*, são defnidos valores padrões para cada um dos seus casos e, ao imprimir o *enum,* utiliza-se a propriedade `seat` definida na declaração do *enum* e que dá acesso ao valor padrão.

1.11 Funções e closures

1.11.1 Funções

Muitas vezes, ao longo do desenvolvimento de um App, nos deparamos com trechos de funcionalidades que precisam ser reutilizados ao longo do código. Esses blocos de código podem ser criados por meio do uso de funções que, são trechos de comandos que executam operações definidas, podem receber valores (parâmetros) para trabalhar e também podem retornar um resultado.

1.11.2 Criando funções

O Código-fonte “Funções” apresentam exemplos referentes do uso de funções.

```
     //Sintaxe para criação de funções:

fun main(args: Array<String>) {

/*
 fun nomeDaFuncao(parâmetro: Tipo) : TipoDeRetorno {
    //Códigos
     return TipoDeRetorno
 }
*/

//Exemplo de uma função simples que não recebe
//parâmetros e não retorna nada
fun printlnHelloFormal() {
    println("Hello!!!!")
}
printlnHelloFormal()    //Hello!!!

fun printlnHelloModoReduzido() = println("Hello!!!! Modo
reduzido!")

printlnHelloModoReduzido()    //Hello!!!

//Função que aceita parâmetro
fun say(message: String) {
    println(message)
}
say("Vamos criar funções em Kotlin")

//Função que aceita mais de um parâmetro e que retorna algo
fun sumNumbers(a: Int, b: Int) : Int {
```

```
    return a + b
}
var result = sumNumbers(10,15)
println(result)   //15

}
```

Código-fonte 1.33 – Funções
Fonte: Elaborado pelo autor (2019)

Percebe-se que quando uma função é invocada, é necessário escrever os parâmetros na sua chamada, ou seja, ao chamar a função `sumNumbers` foi necessário enviar os valores **10** e **15** para a função processar os códigos. O Código-fonte “Exemplo de função apresentando Fibonacci” mostra o exemplo de uma função que calcula os 10 primeiros números da série de Fibonacci.

```
     //Exemplo de função para calcular os 10
//primeiros números da sequência de
//Fibonacci
fun main(args: Array<String>) {

    //Função
    fun sequenciaFibonacci() {

        //Declaração de variáveis
        var number1 = 0
        var number2 = 1

        //Loop controlado de 1 até 10
        for (sequence in 1..10) {

            //Impressão do conteúdo da variável number1
            println("$sequence -> $number1")

            //Soma dos 2 valores das variáveis
            var sum = number1 + number2

            //Troca os valores entre as variáveis
            number1 = number2
            number2 = sum
        }
    }

    //Executar a função
    sequenciaFibonacci()
}
```

Código-fonte 1.34 – Exemplo de função apresentando Fibonacci
Fonte: Elaborado pelo autor (2019)

1.11.3 Single-Expression functions

É possível desenvolver, na linguagem de programação Kotlin, funções que encapsulam outras funções, como mostra o Código-fonte “Função que retorna outra função”.

```
fun main(args: Array<String>) {

    fun double(x: Int): Int = x * 2

    println(double(8))

    fun triple(x: Int) = x * 3

    println(triple(10))

}
```

Código-fonte 1.35 – Função que retorna outra função
Fonte: Elaborado pelo autor (2019)

1.12 Map, Filter e Reduce

Muitas vezes, é necessário efetuar operações em *Arrays* ou Coleções em que é necessário percorrer a coleção para extrair determinados elementos ou apenas modificá-los. Normalmente, a primeira ideia à mente é utilizar o **for in** para iterar a coleção e realizarmos o desejado. Porém existem vários métodos que utilizam *closures* para esses fins. Veja o exemplo “Resumo de Map, Filter e Reduce”.

```
     fun main(args: Array<String>) {

    //elaborando um coleção do tipo List com números entre 1
a 10
    val numbers = listOf(1,2,3,4,5,6,7,8,9,10)
    println(numbers)

    //"Filtrando"(filter) somente os números pares da
coleção numbers
    //a variável temporária chamada "it" utilizada na
operação
    var evenNumbers = numbers.filter { it%2 == 0 }
    println("Listagem de números Pares: $evenNumbers")

    //"Filtrando"(filter) somente os números ímpares da
coleção numbers
    //a variável temporária chamada "it" utilizada na
operação
    var oddNumbers = numbers.filter { it%2 != 0 }
    println("Listagem de números Ímpares: $oddNumbers")

    //A utilização do Map executa o processamento individual
    //de cada elemento dentro da coleção.
    var multiplyNumbers = numbers.map { it * it }
    println("Multiplicação: $multiplyNumbers")

    //Executa o processamento da coleção de acordo
    //com os paramentros enviados.
    var sumNumbers = numbers.reduce {
        //Captura o valor anterior ou atual (acc) e o valor
atual(it)
        acc, it ->
        //Apresenta as informações
        println("acc = $acc, it = $it")
        //Executa o processamento das informações
        acc + it }

    println("Resultado da Somatório: $sumNumbers") //Total
1+2+3+4+5+6+7+8+9+10=55

}
```

Código-fonte 1.36 – Resumo de Map, Filter e Reduce
Fonte: Elaborado pelo autor (2019)

O **map** é um método presente em coleções que percorre a coleção e executa uma *closure* em cada um de seus elementos, devolvendo a nova coleção gerada. O Código-fonte “Map” mostra um exemplo disso.

```
     fun main(args: Array<String>) {

    var names = arrayOf("João", "Paulo", "Henrique", "Ana",
"Beatriz", "Carla", "Caroline")

    //Aplicando map em names
    var uppercasedNames = names.map({it.toUpperCase()})
    println(uppercasedNames)

   //["JOÃO", "PAULO", "HENRIQUE", "ANA", "BEATRIZ",
"CARLA", "CAROLINE"]

}
```

Código-fonte 1.37 – Map
Fonte: Elaborado pelo autor (2019)

Foi criada uma *closure* que retorna a versão em maiúsculas (usando o método de *String toUpperCase**()***) de todos os nomes. O **"it"**, nesse caso, se refere ao parâmetro da *closure* que representa cada um dos nomes da coleção.

No próximo exemplo, o desejo é filtrar a nossa coleção e gerar um novo *Array* contendo apenas os nomes compostos por *cinco* letras ou menos, ou seja, agora será criado um novo *Array* contendo parte dos elementos do *Array* principal. Quando existem casos como esse, deve ser utilizado o método **filter** que, como o próprio nome sugere, filtra uma coleção, devolvendo outra com os elementos que foram filtrados.

```
    //Aplicando filter em names
var filteredNames = names.filter({it.length < 6})
println(filteredNames)
//["João", "Paulo", "Ana", "Carla"]
```

Código-fonte 1.38 – Filter
Fonte: Elaborado pelo autor (2019)

O método filter solicita que seja passada a função que servirá para filtrar os elementos. Essa função deve conter a lógica que será implementada em cada um dos elementos do *Array* e, caso essa lógica retorne **true,** aquele elemento deverá fazer parte do novo *Array*. Veja que, nesse exemplo, foi criada uma *closure* que verifica se a contagem de caracteres de cada nome é menor que **seis**, ou seja, se tem **cinco** ou menos letras.

No último exemplo, será criado um *Array* de **Double** que representa algumas movimentações feitas em uma conta corrente, ou seja, entrada e saída de valores (valores positivos representam entrada, negativos, saída). Como se pode saber o saldo final dessa conta?

Existe um método em Kotlin que está presente em todas as coleções e que serve para combinar, ou seja, juntar todos os valores presentes naquela coleção, segundo uma lógica estipulada por nós. Neste exemplo, essa combinação deverá ser feita por meio da soma de todos esses valores, porém é permitido implementar a lógica que desejar. Vale ressaltar que, apesar de o exemplo utilizar o Double, pode-se utilizar Arrays de qualquer tipo, alterando a lógica a ser implementada. Esse método é o **reduce**, e vamos ver, no Código-fonte “Reduce”, como ficaria sua implementação.

```
     fun main(args: Array<String>) {

    //Utilizando Reduce
    var transactions = arrayOf<Double>
        (500.0, -45.0, -70.0, -25.80, -321.72, 190.0, -35.15, -
100.0)

    var balance = transactions.reduce {

        acc,it -> println("Saldo: " + String.format("%.2f",
acc) +
                                " => Próximo Lançamento: " +
String.format("%.2f", it))
        (acc + it)
    }

    println("Seu saldo é R$ " + String.format("%.2f", balance))
    //Seu saldo é R$ 92,33

}
```

Código-fonte 1.39 – Reduce
Fonte: Elaborado pelo autor (2019)

O método **reduce** é um método que recebe dois parâmetros. O primeiro, chamado **acc,** contém o valor inicial da operação e o segundo, **it,** contém uma *closure* que receberá, a cada iteração, o resultado da operação e o elemento do *Array*. Nesse exemplo, é definido que o valor inicial seria **0** e que, a cada iteração, o valor será somando com o elemento do *Array*, ou seja, na primeira iteração teremos **0.0** + **500.0,**

na segunda, **500.0** + -**45.0,** na terceira, **455.0 +** -**70.0,** e assim sucessivamente, até chegar ao último elemento do *Array.*

1.13 Generics

Você deve ter notado, na sintaxe do método *reduce,* um tipo que não conhecemos, o tipo **"it".** Esse não é um tipo válido ou existente na linguagem, mas foi criado na assinatura desse método e serve para representar um tipo genérico, ou seja, uma indicação de que qualquer tipo pode ser utilizado naquele parâmetro. Perceba que o método exige que o mesmo tipo utilizado no parâmetro **"it"** seja usado dentro da *closure* e também seja o tipo de retorno da função. Esse tipo foi definido logo após o nome do método (reduce**<it>**) e isso indica que, nesse método, será usado um tipo chamado de **"it"**, que pode ser representado por qualquer tipo existente ou criado por você.

O nome deste recurso é **Generics**, e é um recurso poderoso em linguagens de programação, pois com ele não ficamos limitados a um tipo específico quando são criados métodos ou classes. Para exemplificar seu uso, imagine que o gerente do nosso projeto pediu para ser criada uma função que receba dois números inteiros e retorne os mesmos dois números, mas com as posições trocadas. Passadas duas semanas, ele pede para criarmos a mesma função, mas agora temos de trocar duas Strings. Notaram o problema? Sempre teremos que criar novas funções caso o tipo do parâmetro mude. Para isso, podemos criar uma única função com o recurso Generics, como é mostrado no Código-fonte "Generics".

```
    fun main() {

        //Função para trocar números inteiros
        fun swapInt(num1: Int, num2: Int): Pair<Int, Int> {
            return Pair(num2, num1)
        }
        //Função para trocar String
        fun swapString(string1: String, string2: String):
Pair<String, String> {
            return Pair(string2, string1)
        }

        //Resultado
```

```
        println(swapInt(4,400))
        //println(swapInt("TEST1", "TEST2")) //ERRO
        println(swapString("TEST1","TEST2"))
        //Função para trocar qualquer elemento
        fun<T>swapAnything(element1: T, element2: T):
Pair<T, T> {
           return Pair(element2, element1)
        }


         println(swapAnything(4, 400)) //(400, 4)
         println(swapAnything("Test1", "Test2")) //(Test2,
Test1)
         println(swapAnything(20.5, 32.5)) // (32.5, 20.5)


    }
```

Código-fonte 1.40 – Generics
Fonte: Elaborado pelo autor (2019)

1.14 Classes

1.14.1 Definição e construção

Em relação ao uso de classes em Kotlin, é importante ressaltar as seguintes características:

- Definem propriedades para armazenar valores.
- Definem métodos para fornecer funcionalidades.
- Definem inicializadores para configurar seu estado inicial.
- Podem ser estendidas para expandir suas funcionalidades além das presentes nas suas implementações.
- Trabalham com herança, o que permite que uma classe possa herdar as características de outra.
- Type casting, que permite que você possa checar e interpretar uma classe como sendo outra.

Para criar uma classe, deve ser utilizada a palavra reservada **class**, seguida do nome da classe (iniciando em maiúsculo) e sua implementação entre chaves (**{ }**).

Veja o exemplo de criação de uma classe chamada **Person** no Código-fonte "Classes"**.**

```
     class Person constructor(var name: String, var isMale:
Boolean, var age: Int = 0) {

    //Métodos de classe
    fun speak(sentence: String) {
        if (age < 3) {
            println("gugu dada")
        } else {
            println(sentence)
        }
    }

    fun introduce() = println("\nOlá, meu nome é $name e
tenho $age anos de idade.")
}

fun main(args: Array<String>) {

    //Instanciando a classe Person
    var pac = Person("Pedro Alvares Cabral", true)

    //Impressão dos valores antes de alterar a idade
    pac.introduce()

    //Alterando uma propriedade de pac
    pac.age = 45

    //Impressão dos valores depois de alterar a idade
    pac.introduce()

    //Utilizando o método speak
    pac.speak("Treinamento Kotlin")
}
```

Código-fonte 1.41 – Classes
Fonte: Elaborado pelo autor (2019)

A classe **Person** representa uma pessoa e possui as propriedades *name, isMale e age*, que armazenam o nome, sexo (se for **true** significa que é masculino) e a idade. Em uma classe, as propriedades que armazenam um conteúdo também são chamadas de propriedades armazenadas.

Toda classe necessita de um **método construtor** (ou método inicializador) para criar uma instância daquela classe (também chamado de objeto).

DICA: Dentro de classes, as funções passam a ser chamadas de métodos, e as variáveis são chamadas de propriedades.

O **método construtor** é o método que cria uma instância daquela classe e tem por obrigação alimentar qualquer propriedade que não tenha sido inicializada. No exemplo desenvolvido, a propriedade age é a única que foi definida e já inicializada com um valor **(0).** As demais (*name e isMale*) precisam ser inicializadas, e cabe ao método construtor efetuar esta tarefa, e é por isso que o construtor solicita dois parâmetros, **name e isMale,** que serão repassados às respectivas propriedades. Vale ressaltar que o nome dos parâmetros não precisa, necessariamente, ser o mesmo.

A nossa classe **Person** possui dois métodos, o **introduce()**, que serve para retornar a apresentação da pessoa, e **speak(sentence: String)**, que faz com que a pessoa fale algo. Nesse método, será incluída uma regra informando que uma pessoa com menos de **três** anos só fala **"gugu dada"**.

Para instanciarmos uma classe, criamos uma variável e atribuímos a ela a chamada do método construtor da classe que desejamos. Para chamarmos o método construtor, usamos apenas o nome da classe, passando os valores dos parâmetros do método.

1.14.2 Propriedades computadas

A classe desenvolvida possui três propriedades armazenadas. Porém, em Kotlin, pode-se fazer uso do que chamamos de propriedade computada. Imagine que, em nossa classe, fosse interessante ter uma propriedade chamada *gender*, que retornaria uma *String* contendo "masculino" ou "feminino", de acordo com o sexo da pessoa. Poderíamos, tranquilamente, criar essa propriedade, porém concorda que essa informação pode ser recuperada a partir de **isMale,** e que criar uma nova propriedade resultaria em alimentar duas propriedades para termos a mesma informação? Uma das soluções para esse problema seria criarmos um método que retornasse o sexo em *String*, porém métodos são recursos que fazem mais sentido quando precisamos passar um parâmetro para obter um resultado e, no nosso caso, *gender* é uma definição de **Person,** é uma característica, e características são representadas por propriedades.

Existe um recurso em Kotlin que nos permite ter uma propriedade que não armazena nenhum valor, apenas utiliza e trabalha um valor existente, a essas propriedades chamamos propriedades computadas. Vejamos como poderíamos criar a propriedade computada *gender* no Código-fonte “Propriedades computadas”.

```
     class Person {

         //.....

//Propriedade computada
val gender: String
    get() {
    if (isMale) {
        return "masculino"
    } else {
        return "feminino"
    }
}

       //.....
     }
```

Código-fonte 1.42 – Propriedades computadas
Fonte: Elaborado pelo autor (2019)

Agora, caso seja desejado imprimir o sexo da pessoa, pode-se utilizar *gender* em vez de *isMale*, pois fica mais fácil e legível ao usuário visualizar o sexo como **“masculino”** e **“feminino”** do que como **true** e **false**.

1.14.3 Propriedades/métodos de classe

Todas as propriedades criadas na classe **Person** são chamadas de **propriedades de instância**, o que significa que seu uso só é possível por meio de uma instância da classe (através de um objeto). Podemos criar propriedades que não necessitem de uma instância para serem utilizadas e que possam ser acessadas diretamente na classe. Tais propriedades são chamadas de **propriedades de classe**.

Na classe **Person**, será criada uma propriedade de classe que retorna a classe de animal da qual uma pessoa faz parte (mamífero). Como essa é uma informação referente à própria classe em si, ou seja, toda pessoa é um mamífero, deverá ser criada como propriedade de classe, pois, assim, pode-se saber a qual classe animal uma **Person** pertence, sem a necessidade de criarmos uma instância dessa classe.

Propriedades de classe são criadas utilizando as palavras reservadas **companion object** e, por isso, costumam ser chamadas também de propriedades estáticas. Uma propriedade estática mantém seu valor, se alterado ao longo do código, o que a torna útil em determinados cenários. Além de propriedades de classe, também pode haver métodos de classe que, como as propriedades, podem ser utilizados sem a necessidade de uma instância. O Código-fonte “Propriedades e métodos de classe” apresenta como criar métodos e propriedades de classe em Kotlin. Tais propriedades e métodos serão inseridos na classe **Person.**

```
     class Person constructor(var name: String, var isMale:
Boolean, var age: Int = 0) {

    //Métodos de classe
    fun speak(sentence: String) {
        if (age < 3) {
            println("gugu dada")
        } else {
            println(sentence)
        }
    }

    fun introduce() = println("\nOlá, meu nome é $name e
tenho $age anos de idade.")

    //Propriedade computada
    val gender: String
        get() {
        if (isMale) {
            return "masculino"
        } else {
            return "feminino"
        }
    }

         //palavras reservadas dentro da classe
    //que habilitam propriedades e métodos
    //que podem ser acessados diretamente.
    companion object {

        //Propriedade de classe (estática)
        var animalClass: String = "mamífero"

        //Método de classe
        fun getInfo() : String {
            return "Pessoa: ${Person.animalClass} que possui
nome, sexo e idade"
        }
```

```
    }
}

fun main(args: Array<String>) {


    //Instanciando a classe Person
    var pac = Person("Pedro Alvares Cabral", true)

    //Impressão dos valores antes de alterar a idade
    pac.introduce()

    //Alterando uma propriedade de pac
    pac.age = 45

    //Impressão dos valores depois de alterar a idade
    pac.introduce()

    //Utilizando o método speak
    pac.speak("Treinamento Kotlin")

    println(pac.gender)

    println(Person.animalClass) //mamífero

    println(Person.getInfo())
    //Pessoa: mamífero que possui nome, sexo e idade
}
```

Código-fonte 1.43 – Propriedades e métodos de classe
Fonte: Elaborado pelo autor (2019)

Um método ou uma propriedade de classe são utilizados por meio da própria classe em si, e não da instância. É por isso que necessita chamá-lo na própria classe. Até mesmo internamente — como no caso do método **getInfo()** — é necessário referenciá-lo por meio da classe.

1.14.4 Herança

Uma das principais vantagens de se trabalhar com classes é que elas fazem uso de herança, ou seja, podem herdar as características de outra classe que, nesse caso, chamamos de **classe mãe** ou **super**.

Em Kotlin, para definirmos que uma classe herda de outra, é utilizado o sinal **:** (dois pontos) após o seu nome, seguido da classe da qual herdará. Caso a classe filha implemente uma nova propriedade e essa não tenha nenhum valor associado

durante sua definição, deve-se criar um novo construtor que terá o papel de alimentar tanto essa quanto todas as propriedades não inicializadas da classe mãe. Nesse caso, deve-se, primeiro, inicializar as propriedades da classe **filha** para depois inicializar as propriedades da classe **mãe,** e essa etapa é feita chamando o construtor da classe **super**, usando a palavra reservada **super.**

A próxima etapa é construir uma classe chamada **Student** que irá representar um estudante e, como todo estudante é uma pessoa, será criada herdando da classe **Person**. O estudante em questão possui uma propriedade a mais, o **rm** e, em função disso, necessitará de um construtor *próprio.* Veja a implementação no Código-fonte "Herança".

```
     open class Person constructor(var name: String, var
isMale: Boolean, var age: Int = 0) {

    //Métodos de classe
    fun speak(sentence: String) {
        if (age < 3) {
            println("gugu dada")
        } else {
            println(sentence)
        }
    }

    fun introduce() = println("\nOlá, meu nome é $name e
tenho $age anos de idade.")

    //Propriedade computada
    val gender: String
        get() {
        if (isMale) {
            return "masculino"
        } else {
            return "feminino"
        }
    }

    //palavras reservadas dentro da classe
    //que habilitam propriedades e métodos
    //que podem ser acessados diretamente.
    companion object {

        //Propriedade de classe (estática)
        var animalClass: String = "mamífero"
```

```
        //Método de classe
        fun getInfo() : String {
            return "Pessoa: ${Person.animalClass} que possui
nome, sexo e idade"
        }

    }

}

//HERANÇA
class Student : Person {
    constructor (name: String, isMale: Boolean, age: Int = 0,
rm: String) : super(name,isMale,age) {
    }
}


fun main(args: Array<String>) {


    //Instanciando a classe Person
    var pac = Person("Pedro Alvares Cabral", true)

    //Impressão dos valores antes de alterar a idade
    pac.introduce()

    //Alterando uma propriedade de pac
    pac.age = 45

    //Impressão dos valores depois de alterar a idade
    pac.introduce()

    //Utilizando o método speak
    pac.speak("Treinamento Kotlin")

    println(pac.gender)

    println(Person.animalClass) //mamífero

    println(Person.getInfo())
    //Pessoa: mamífero que possui nome, sexo e idade

    var student = Student("Pedrinho Cabral", false,
10,"97663")
    student.introduce();
}
```

Código-fonte 1.44 – Herança
Fonte: Elaborado pelo autor (2019)

Observe que o método *construtor* de **Student** precisa solicitar todas as informações para não só criar um *Student*, como também criar um **Person**, cujo construtor é chamado pelo uso de **super**, que nos dá acesso direto à classe **mãe**.

1.14.5 Sobrescrita

As classes **filhas** podem modificar propriedades ou métodos das classes **mãe,** utilizando uma técnica chamada sobrescrita (*override*) Em Kotlin, é possível fazer isso com a palavra reservada **override** seguida da nova implementação do método ou propriedade que será modificado. Com essa técnica, uma classe não precisa, necessariamente, executar os métodos ou propriedades computadas da mesma forma que a sua classe **mãe** executa.

A modificação proposta para o método **introduce()** faz com que agora, ele, além de retornar a apresentação do estudante pelo seu nome, também informe o seu *RM,* e faremos uso do próprio método *introduce()* da classe **mãe** para recuperar essa informação inicial. As classes **Person** e **Student** ficarão assim:

```
     open class Person constructor(var name: String, var
isMale: Boolean, var age: Int = 0) {

    //Métodos de classe
    fun speak(sentence: String) {
        if (age < 3) {
            println("gugu dada")
        } else {
            println(sentence)
        }
    }

    open fun introduce() = println("\nOlá, meu nome é $name e
tenho $age anos de idade.")

    //Propriedade computada
    val gender: String
        get() {
        if (isMale) {
            return "masculino"
        } else {
            return "feminino"
        }
    }
```

```
    //palavras reservadas dentro da classe
    //que habilitam propriedades e métodos
    //que podem ser acessados diretamente.
    companion object {

        //Propriedade de classe (estática)
        var animalClass: String = "mamífero"

        //Método de classe
        fun getInfo() : String {
            return "Pessoa: ${Person.animalClass} que possui
nome, sexo e idade"
        }

    }

}

//HERANÇA
class Student : Person {

    //propriedade local da classe Student
    var rm = String()

    constructor (name: String, isMale: Boolean, age: Int = 0,
rm: String) : super(name, isMale, age) {
        //Atribuindo o valor na propriedade local da classe
Student
        this.rm = rm
    }

    override fun introduce(): Unit {
        super.introduce()
        //Acessando a informação da propriedade local da
classe Student
        println("meu RM nesta escola é $rm")
    }
}


fun main(args: Array<String>) {


    //Instanciando a classe Person
    var pac = Person("Pedro Alvares Cabral", true)

    //Impressão dos valores antes de alterar a idade
    pac.introduce()

    //Alterando uma propriedade de pac
```

```
    pac.age = 45

    //Impressão dos valores depois de alterar a idade
    pac.introduce()

    //Utilizando o método speak
    pac.speak("Treinamento Kotlin")

    println(pac.gender)

    println(Person.animalClass) //mamífero

    println(Person.getInfo())
    //Pessoa: mamífero que possui nome, sexo e idade

    var student = Student("Pedro Júnior ", false, 10,"97663")
    student.introduce();
}
```

Código-fonte 1.45 – Sobrescrita
Fonte: Elaborado pelo autor (2019)

1.15 Considerações sobre a Introdução ao Kotlin

Neste capítulo, foi possível conhecer essa linguagem adotada para o desenvolvimento de aplicativos para os diferentes gadgets que suportam o sistema operacional Android. Aprender Kotlin é o pontapé inicial e fundamental para entrar no mundo de desenvolvimento de Apps para as plataformas do Google, e, por meio deste módulo, foi possível conhecer, treinar e aprender como utilizar os principais recursos dessa linguagem, desde a simples criação de variáveis até o uso de recursos avançados.

REFERÊNCIAS

KOTLIN. 2020. Disponível em: <https://kotlinlang.org/>. Acesso em: 05 out. 2020.

VIANA, D. **Kotlin**: a nova linguagem oficial para desenvolvimento Android. 2017. Disponível em: <https://www.treinaweb.com.br/blog/kotlin-a-nova-linguagem-oficial-para-desenvolvimento-android/>. Acesso em: 05 out. 2020.